Isolation levels Recovery and ARIES

Where we are

- Lots of discussion on Isolation
- Definitions
- Locking/nonlocking protocols

Reading

- [RG] 16.6, 16.7, 18.1-18.6

Isolation Levels

- In some cases, enforcing full serializability can cause a significant performance hit
- This idea led to isolation levels:
  - Relax isolation and admit some anomalies for better performance
- The SQL standard provides four isolation levels

SQL isolation levels

- Based on the fundamental isolation anomalies
  - Dirty reads
  - Unrepeatable reads
  - Phantoms (phantom reads)
  - (Lost updates)

The SQL Isolation Levels

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

The SQL Isolation Levels

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

- Isolation level can be set per transaction

Isolation Levels

- READ UNCOMMITTED
  - Dirty reads, unrepeatable reads and phantoms can occur
  - Very few guarantees
  - In real systems, transactions are typically restricted to be read-only
    - That ensures no lost updates
    - But it's not a requirement in the definition of READ UNCOMMITTED

Isolation Levels

- READ COMMITTED
  - No lost updates and no dirty reads
  - Unrepeatable reads and phantoms can still occur
  - Locking implementation: hold write locks until commit, release read locks early

Isolation Levels

- REPEATABLE READ
  - Reads are now repeatable as well
  - Locking implementation: Read and write locks held until commit
  - Phantoms can still happen

Isolation Levels

- SERIALIZABLE
  - No phantoms permitted either
  - Predicate locks or similar mechanisms must be used

Isolation Level Summary

Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	No	Possible	Possible
REPEATABLE READ	No	No	Possible
SERIALIZABLE	No	No	No

Are these really used?

- Yes, a lot!
- In fact, the default isolation level may not be SERIALIZABLE if you install and use a system "out of the box"
- READ COMMITTED quite popular

Serializable vs. serializability

- Last time: snapshot isolation
- Provides **`SERIALIZABLE`** isolation
  - Dirty reads?
  - Unrepeatable reads?
  - Phantom reads?
- This is what some DBMS's (e.g. Oracle) use as their implementation of **`SERIALIZABLE`**

But, SI allows write skew anomalies

```
create table a ( x int );
create table b ( x int );
```

t	Transaction T1	Transaction T2
1	`Insert into a select count(*) from b;`	
2		`Insert into b select count(*) from a;`
3	`Commit;`	
4		`Commit;`

Moral of the story

- Gap between the standard (SERIALIZABLE) and the theoretical concept
- Snapshot isolation is a valid implementation of the standard
- You need to decide what is right for your application

Isolation summary

- Correctness criteria based on serializability (and abort handling)
- Protocols:
  - Locking-based
  - Nonlocking
- Isolation levels
- Limited (though growing) support in NoSQL systems

Review: The ACID properties

- **A**tomicity: All actions in a transaction happen, or none happen
- **C**onsistency: Each transaction transforms the database from one consistent state to another
- **I**solation: Execution of concurrent transactions is as though they are evaluated in some serial order
- **D**urability: If a transaction commits, its effects persist

The **Recovery Manager** guarantees Atomicity & Durability.

Atomicity

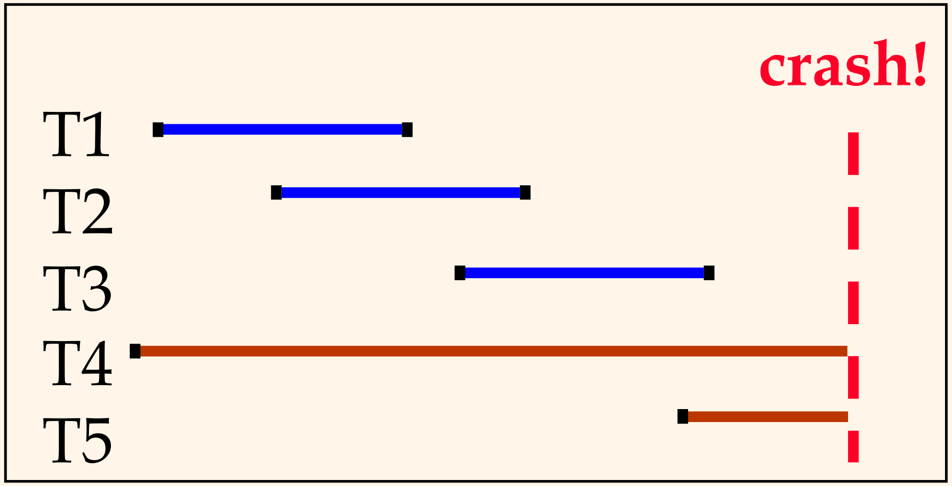
- If a transaction can't complete and aborts, system must undo its changes completely
  - Abort could happen because transaction decides to abort
  - or system may need to abort transaction for some reason (e.g. to break a deadlock)
  - or a transaction could be "in-flight" at the moment of a crash and we decide to undo its effects

Durability

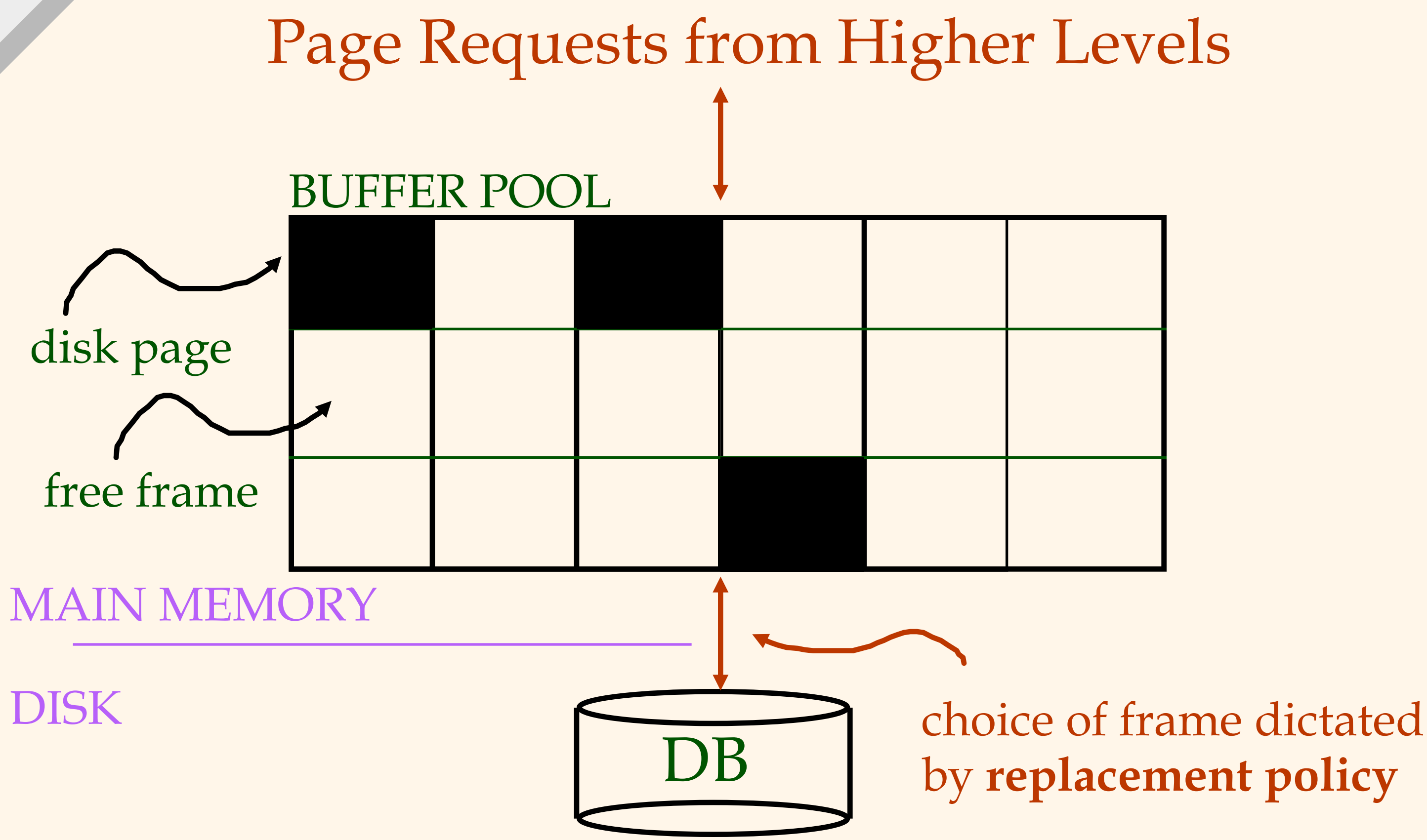
- If a transaction commits, system must preserve its effects (writes)
  - Even if the system crashes before those writes make it to disk
  - In such a case, system needs ability to redo writes

Desired behavior after crash

- T1, T2 & T3 should be durable.
- T4 & T5 should be aborted (effects not seen).



Buffer Management in a DBMS



Handling the Buffer Pool

- Force every write to disk when a transaction commits?
  - If yes, poor response time.
- Steal buffer-pool frames from uncommited transactions?
  - If not, poor throughput.

	No Steal	Steal
Force		
No Force		**Desired**

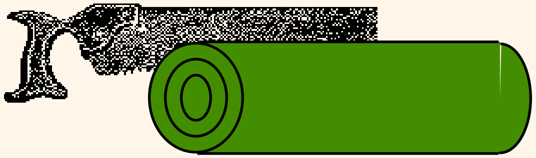
STEAL and atomicity

- *To steal frame F:* Current page in F (say P) is written to disk; some transaction has made changes to P.
- What if the transaction then aborts?
- Must remember the old value of P at steal time (to support UNDOing the write).

NO FORCE and Durability

- What if system crashes before a modified page is written to disk?
- Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.

Basic Idea: Logging



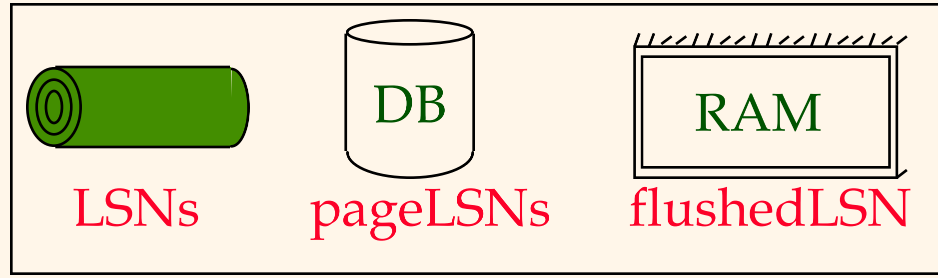
- Record REDO and UNDO information, for every update, in a *log* that will survive crashes.
  - Log is written sequentially.
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- Log: An ordered list of REDO/UNDO actions
  - Log record contains:

    <transID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).

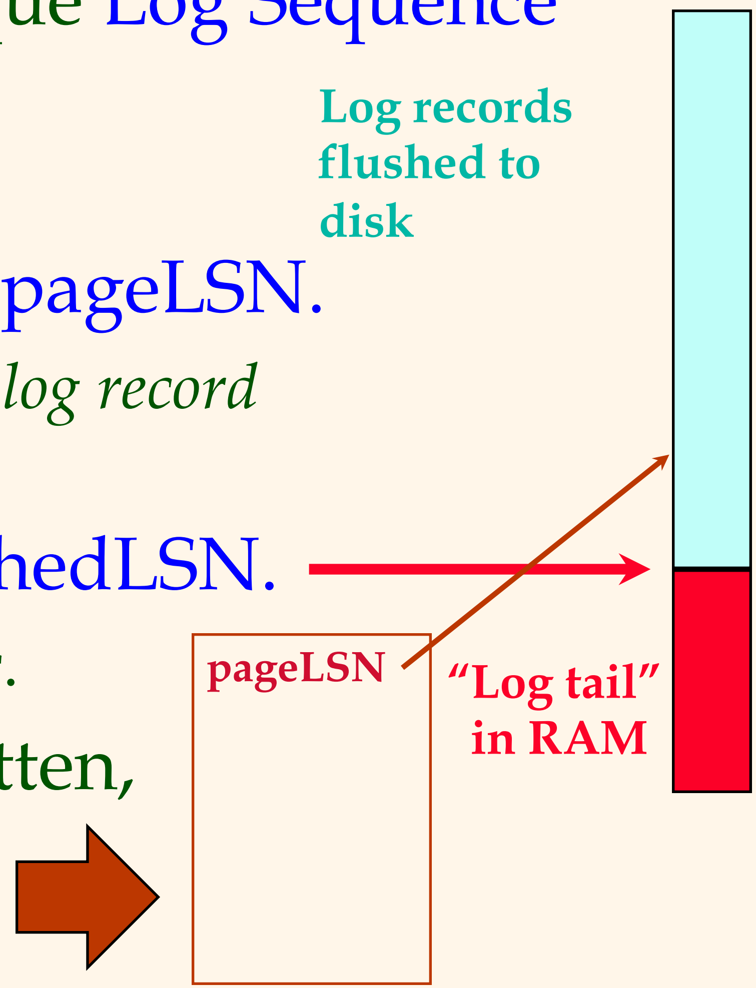
Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  - Must force the log record for an update before the corresponding data page gets to disk.
  - Must write all log records for a transaction before commit.
- #1 guarantees Atomicity (why?)
- #2 guarantees Durability (why?)
- Exactly how is logging (and recovery!) done?
  - We'll study the ARIES algorithm.

WAL & the Log



- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each data page contains a pageLSN.
  - The LSN of the most recent *log record* for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: *Before* a page is written,
  - pageLSN ≤ flushedLSN



Log Records

Possible log record types:

- Update
- Commit
- Abort
- End
  - signifies end of cleanup after commit or abort
- Compensation Log Records (CLRs)
  - for UNDO actions

Log Records

LogRecord fields:

prevLSN
transID
type
pageID
length
offset
before-image
after-image

update records only

Other ARIES-Related State

- Transaction Table:
  - One entry per active transaction.
  - Contains transID, status (running/commited/aborted), and lastLSN.
- Dirty Page Table:
  - One entry per dirty page in buffer pool.
  - Contains recLSN -- the LSN of the log record which ***first*** caused the page to be dirty.

The Big Picture: What's Stored Where

LogRecords

prevLSN
transID
type
pageID
length
offset
before-image
after-image

Data pages

each
with a
pageLSN

Transaction Table

lastLSN
status

Dirty Page Table

recLSN

flushedLSN

Normal Execution of a transaction

- Series of reads & writes, followed by commit or abort.
  - We will assume that write is atomic on disk.
    - In practice, additional details to deal with non-atomic writes.
- Strict 2PL → very important!
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.

Checkpointing

- Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash.
- A checkpoint is NOT a snapshot of the whole DB state
  - It is much more lightweight
  - Only record the transaction table and dirty page table

Checkpointing

- To take a checkpoint, write to log:
  - begin_checkpoint record: Indicates when chkpt began.
  - end_checkpoint record: Contains current *transaction table* and *dirty page table*. This is a fuzzy checkpoint:
    - Other transaction continue to run; so these tables accurate only as of the time of the begin_checkpoint record.
    - No attempt to force dirty pages to disk
- Store LSN of chkpt record in a safe place (master record).

The Big Picture: What's Stored Where

LogRecords

prevLSN
transID
type
pageID
length
offset
before-image
after-image

Data pages

each
with a
pageLSN

master record

Transaction Table

lastLSN
status

Dirty Page Table

recLSN

flushedLSN

Transaction Commit

- Write commit record to log.
- All log records up to Xact's lastLSN are flushed.
  - Guarantees that flushedLSN ≥ lastLSN.
- Commit() returns.
- Remove transaction from transaction table
- Write end record to log

Simple Transaction Abort

- For now, consider an explicit abort.
  - No crash involved.
- We want to "play back" the log in reverse order, UNDOing updates.
  - Get lastLSN of transaction from transaction table.
  - Can follow chain of log records backward via the prevLSN field.
  - Before starting UNDO, write an *Abort* log record.
    - For recovering from crash during UNDO!

Abort, cont.

- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we're currently undoing).
  - CLRs *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an "end" log record.

Example

- 10 T1 writes P5
- 20 T2 writes P17
- 30 T1 writes P3

Frame Steal - P3 written to disk by BM (pageLSN for page 30 at this time is 30, so log must be flushed up to entry 30)

- 40 abort T1
- 50 CLR T1 P3 (undonextLSN: 10)
- 60 CLR T1 P5 (undonextLSN: NULL)
- 70 End T1